LAB. TESTAREA UNITARA INTRODUCERE

Ce este unitate de testare:

1. Testarea funcțiilor individuale ale sistemului.
2. Ca regulă, se efectuează dezvoltarea modulului.
3. Poate fi ușor automatizat.
4. Pune bazele pentru aplicarea testării de regresie.

JUnit - bibliotecă care permite testarea unitara a aplicațiilor Java.

Testele scrise stabilizează și permite reducerea timpul de depanare a codul programului. Testarea sistemului ca un întreg (sistem de testare), nu este întotdeauna posibil să detecteze erorile din componentele individuale. Corectarea erorilor în stadiile incipiente de dezvoltare sunt mai puțin costisitoare.

EXEMPLU

Să presupunem că există o clasă care implementează o serie de funcții matematice:

```
public class CustomMath {
public static int sum(int x, int y){
	return x+y; //returneaza rezultatul adunării a 2 numere
}
public static int division(int x, int y){
	if (y==0) {
		throw new IllegalArgumentException("devider is 0");
// arunca o exceptie
	}
	return(x/y); // returneaza rezultatul impartirii
}
}
```

OBSERVAȚIE

Uneori este necesar să furnizam programului teste unitare. Aceste teste unitare sunt incomode de a le stoca în codul programului, deoarece duce la:

1. Complica citirea codului.
2. Sunt dificile la pornire.
3. Testele nu sunt relevante pentru logica de afaceri a cererii și ar trebui să fie excluse din produsul final.

Bibliotecă externă conectată la proiect, poate facilita în mare măsură dezvoltarea și întreținerea testelor unitare. Cele mai populare bibliotecă pentru Java este JUnit.

TESTAREA UNITARA FĂRĂ UTILIZAREA BIBLIOTECILOR

Unele testări pot fi amplasate în clasa CustomMath. În continuare vom completa clasa CustomMath:

```
public class CustomMath {
public static int sum(int x, int y){
	return x+y;
}
public static int division(int x, int y){
	if (y==0) {
		throw new IllegalArgumentException("devider is 0");
	}
	return(x/y);
}
public static void main (String[] args){
if (sum(1,3)==4){
//verificam daca la adunarea numerelor 1 cu 3 se returneaza 4
	System.out.println("Test1 passed.");
}
else {
	System.out.println("Test1 failed.");
}
try {
	int z=division(1,0);
	System.out.println("Test3 failed.");
}
catch (IllegalArgumentException e){
//Testul se considera reusit daca incercarea de impartire
//la 0 genereaza exceptia asteptata

	System.out.println("Test3 pased.");
}}}
```

Rezultatul execuției programului este prezentat în următoarea figură:

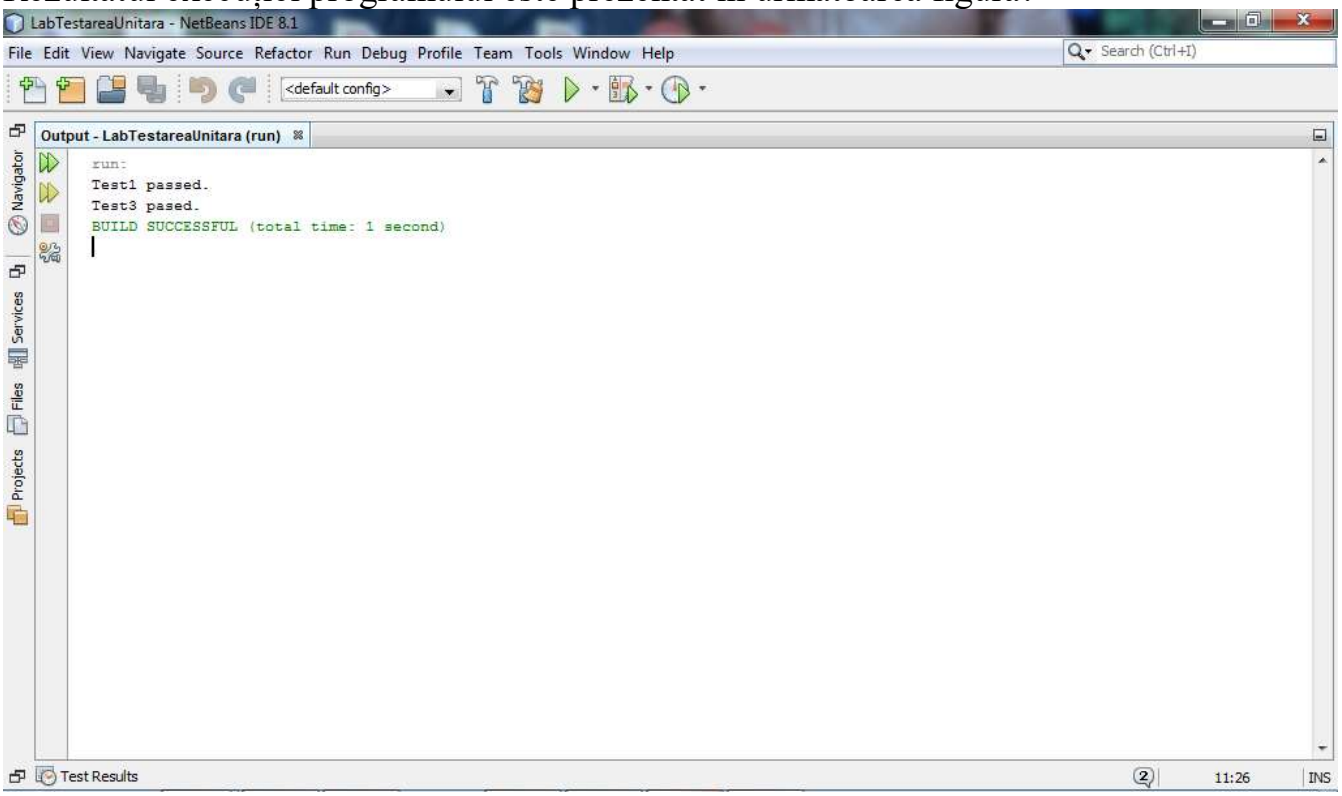

Fig.1 Rezultatul execuției programului

INSTALAREA J-UNIT

JUnit poate fi folosit pentru orice aplicație efectuată în Java. Biblioteca este inclusă în cele mai multe IDE-uri, inclusiv NetBeans.

CREAREA MODULULUI DE TESTARE

Crearea modului de testare după un model poate fi făcută cu ajutorul expertului de testare. Testele JUnit vor fi localizate în cadrul proiectului. Structura mapelor în Test Packages în caz general dublează mapa clasei Source Packages. Alege File-> New File-> ... în conformitate cu JUnit vom alege test pentru o clasă existentă («Test for Existing Class») (Fig.).

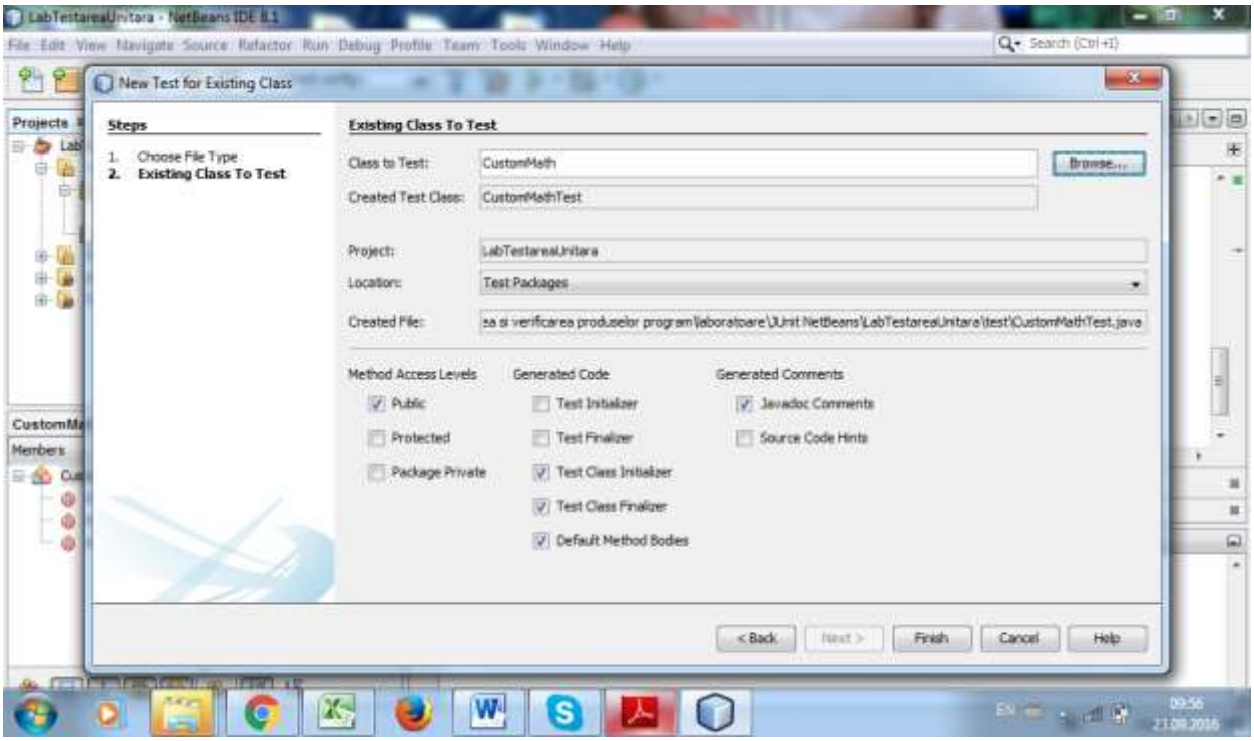

Fig.2 Crearea modului de testare

Javadoc - o formă de organizare în cod a comentariilor folosind cuvinte cheie pentru care NetBeans definește informațiile. În cazul în care clasa este creată folosind Javadoc – atunci poate fi automat creată documentația și de asemenea să funcționeze funcția de asistență sensibilă la context NetBeans (de exemplu, arătarea scopului funcției).

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
```

```
/**
 *
 * @author Andrian
 */
public class CustomMathTest {

    public CustomMathTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    /**
     * Test of sum method, of class CustomMath.
     */
    @Test
    public void testSum() {
        System.out.println("sum");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    /**
     * Test of division method, of class CustomMath.
     */
    @Test
    public void testDivision() {
        System.out.println("division");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.division(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }
    /**
     * Test of main method, of class CustomMath.
     */
    @Test
    public void testMain() {
        System.out.println("main");
        String[] args = null;
        CustomMath.main(args);
        fail("The test case is a prototype.");
    }}
```

În codul testului poate fi văzut rezumatul: Informații cu privire la numirea metodelor cu simbolul @ (@BeforeClass, @AfterClass, @Test).

Notarea @Test arată metode care pornesc automat mediul de testare.

@BeforeClass Și @AfterClass conține pași care trebuie să fie finalizați înainte de a începe clasa testului (de exemplu, conexiunea bazei de date, sau pregătirea datelor pentru prelucrare) sau după executarea testului (de exemplu, deconectarea de la baza de date, a restabili starea inițială).

ÎNCEPEREA CREĂRII TESTĂRII IMPLICITE

Începerea testului are loc prin intermediul meniul contextual al clasei testate, cu opțiunea Test File (Fig.3).

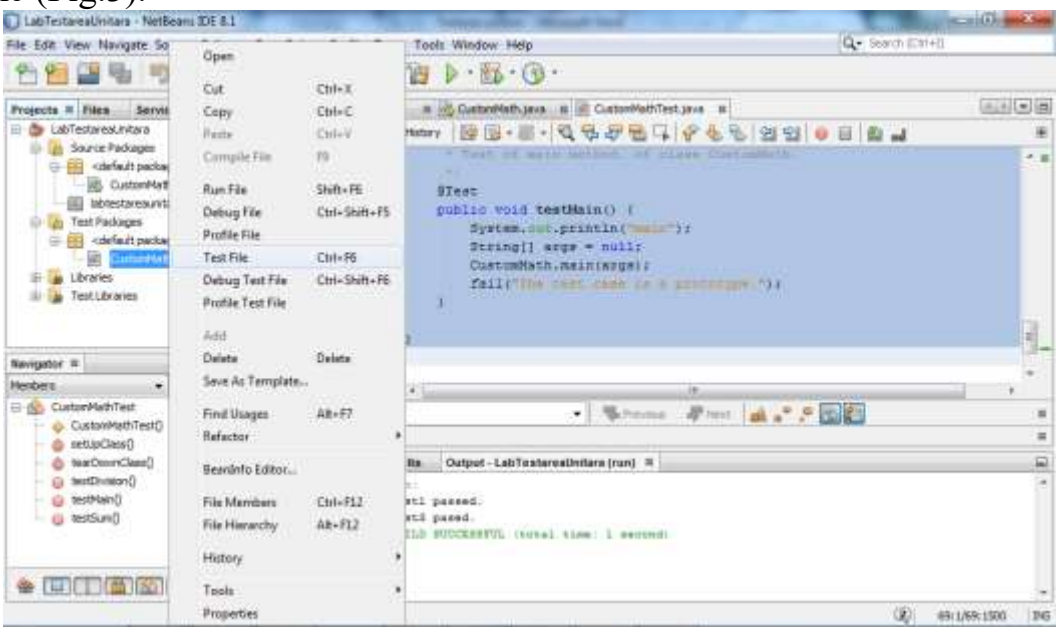

Fig.3 Modul de începere a testului

Fila Test Results afișează mesajele de ieșire la consola, precum și rezultatele testelor privind metodele de testare de clasă(Fig.4).

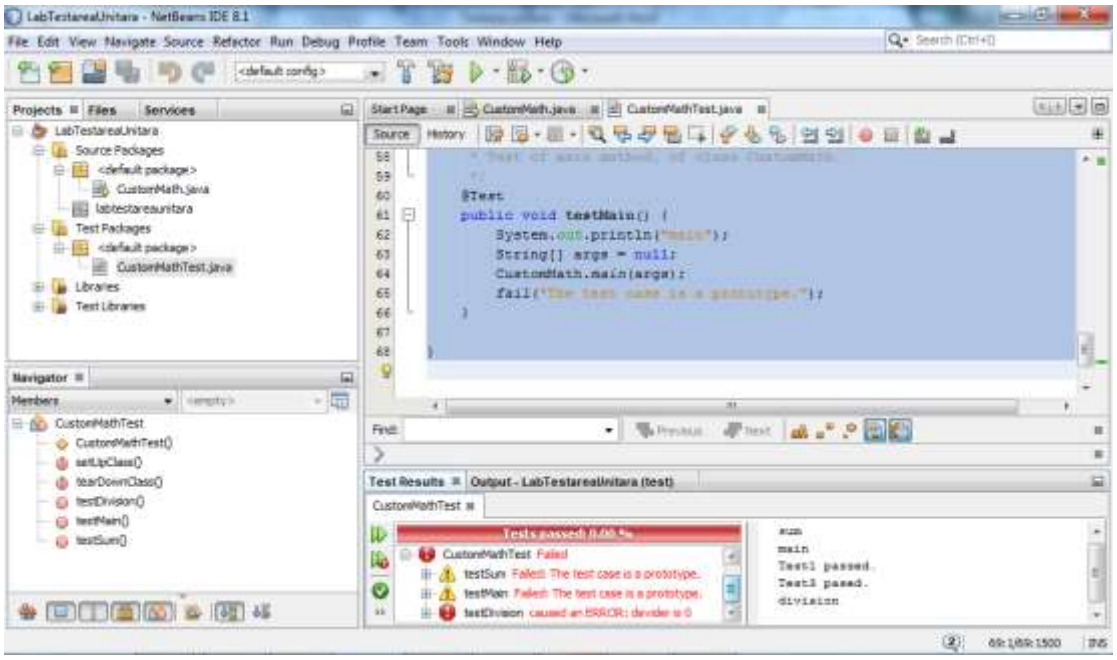

Fig.4 Rezultatul testării

Ierarhia clasei JUnit:

- java.lang.Object
- org.junit.**Assert**
- org.junit.**Assume**
- java.lang.Throwable (implements java.io.Serializable)
- java.lang.Error
- java.lang.AssertionError

org.junit.**ComparisonFailure**

- org.junit.**Test.None**

Annotation Type Hierarchy

- org.junit.**Test** (implements java.lang.annotation.Annotation)
- org.junit.**Ignore** (implements java.lang.annotation.Annotation)
- org.junit.**BeforeClass** (implements java.lang.annotation.Annotation)
- org.junit.**Before** (implements java.lang.annotation.Annotation)
- org.junit.**AfterClass** (implements java.lang.annotation.Annotation)
- org.junit.**After** (implements java.lang.annotation.Annotation)

Pentru a verifica corectitudinea rulării unei metode în grupul JUnit este prevăzut grupul de metode Assert, ce verifică condițiile și în cazul necorespunderii testul nu este trecut.

Descrierea metodelor:

Method Summary

static void	**assertArrayEquals**(byte[] expecteds, [] actuals) Asserts that two byte arrays are l.
static void	**assertArrayEquals**(char[] expecteds, [] actuals) Asserts that two char arrays are l.
static void	**assertArrayEquals**(int[] expecteds,] actuals) Asserts that two int arrays are l.
static void	**assertArrayEquals**(long[] expecteds, [] actuals) Asserts that two long arrays are l.
static void	**assertArrayEquals**(java.lang.Object[] cteds, java.lang.Object[] actuals) rts that two object arrays are equal.
static void	**assertArrayEquals**(short[] expecteds, t[] actuals) Asserts that two short arrays are l.
static void	**assertArrayEquals**(java.lang.String age, byte[] expecteds, byte[] als) Asserts that two byte arrays are equal.
static void	**assertArrayEquals(java.lang.String age, char[] expecteds, char[] als) Asserts that two char arrays are l.**
static void	**assertArrayEquals(java.lang.String age, int[] expecteds, int[] actuals) rts that two int arrays are equal.**

static void	**assertArrayEquals(java.lang.String age, long[] expecteds, long[] als) Asserts that two long arrays are l.**
static void	**assertArrayEquals(java.lang.String age, java.lang.Object[] expecteds, ..lang.Object[] actuals)** Asserts that two object arrays are equal.
static void	**assertArrayEquals**(java.lang.String age, short[] expecteds, short[] als) Asserts that two short arrays are equal.
static void	**assertEquals**(double expected, double al) **Deprecated.** *Use assertEquals(double cted, double actual, double epsilon) ad*
static void	**assertEquals**(double expected, double al, double delta) Asserts that two doubles ats are equal to within a positive delta.
static void	**assertEquals**(long expected, long al) Asserts that two longs are equal.
static void	**assertEquals**(java.lang.Object[] cteds, java.lang.Object[] actuals) **ecated.** *use assertArrayEquals*
static void	**assertEquals**(java.lang.Object cted, java.lang.Object actual) Asserts wo objects are equal.
static void	**assertEquals(java.lang.String age, double expected, double actual) ecated. Use assertEquals(String age, double expected, double actual, le epsilon) instead**
static void	**assertEquals(java.lang.String age, double expected, double actual, le delta) Asserts that two doubles or ts are equal to within a positive a.**
static void	**assertEquals(java.lang.String age, long expected, long actual) rts that two longs are equal.**
static void	**assertEquals(java.lang.String age, java.lang.Object[] expecteds, ..lang.Object[] actuals) Deprecated. assertArrayEquals**
static void	**assertEquals(java.lang.String age, java.lang.Object expected, ..lang.Object actual) Asserts that two cts are equal.**
static void	**assertFalse(boolean condition) rts that a condition is false.**
static void	**assertFalse(java.lang.String message, ean condition) Asserts that a lition is false.**
static void	**assertNotNull(java.lang.Object ct) Asserts that an object isn't .**
static void	**assertNotNull(java.lang.String age, java.lang.Object object) Asserts an object isn't null.**
static void	**assertNotSame(java.lang.Object pected, java.lang.Object actual) rts that two objects do not refer to**

	same object.
static void	**assertNull**(java.lang.Object object) rts that an object is null.
static void	
static void	**assertNull**(java.lang.String message, .lang.Object object) Asserts that an object ll.
static void	**assertSame**(java.lang.Object expected, .lang.Object actual) Asserts that two :ts refer to the same object.
static void	**assertSame**(java.lang.String message, .lang.Object expected, .lang.Object actual) Asserts that two :ts refer to the same object.
static void	**assertNotSame(java.lang.String age, java.lang.Object unexpected, ..lang.Object actual)** Asserts that two :ts do not refer to the same object.
static void	
static void	***fail()*** Fails a test with no message. *fail(java.lang.String message)* Fails st with the given message.

SARCINA 1

Creați un proiect cu clasa CustomMath de mai sus.

Omiteți din metoda main a clasei CustomMath verificarea funcției sum.

Omiteți din metoda testSum apelul metodei fail. Asigurați-vă că testarea funcției sum trece pentru datele de intrare curente.

Adăugați în raport codul funcției testSum și rezultatul testării (rezultatele testelor PrtSc al ferestrei).

TRATAREA EXCEPȚIILOR

Excepțiile pot fi un comportament normal al unei metodei în anumite condiții (de exemplu, excepție în cazul absenței unui fișier). Tratarea excepțiilor într-un test are loc prin intermediul unui bloc try ... catch (), sau să le transmită mai departe cu ajutorul cuvântului cheie throws în corpul metodei.

Vom schimba metoda testDivision, astfel încât să verifice comportamentul corect atunci când are loc împărțirea la 0. Comportamentul corect în acest caz este de a genera o excepție.

SARCINA 2

Modificați testul testDivision în următorul mod:

```
Test

    public void testDivisionByZero() {

        int x = 0;
```

```
        int y = 0;
        int expResult = 0;
        try {
            int result=CustomMath.division(x, y);
            assertEquals(expResult, result);
            if(y==0) fail("Деление на ноли не создает исключителънои ситуации");
            }
        catch(IllegalArgumentException e){
            if(y!=0) fail("Генерация исключения при ненулевом знаменателе");
        }
    }
```

Omiteți verificarea metodei division din metoda main a clasei CustomMath.

Porniți testarea pentru y=0 și y!=0 (manual, schimbînd secvențial valorile inițiale a lui y).

Plasați în raport codul testării și rezultatele testului testDivisionByZero pentru diferite valori ale lui y.

TESTE PARAMETRICE

Pentru a testa logica aplicației este necesar să se creeze teste în mod regulat, numărul testelor poate varia de la caz la caz. În versiunile anterioare ale JUnit acest lucru a dus la inconveniente semnificative - în primul rînd datorită faptului că la schimbarea grupurilor de parametri necesari în metoda testată a dus la scrierea unui test aparte pentru fiecare grup.

JUnit 4 a realizat posibilitatea de a permite crearea testelor comune, care pot trimite o varietate de parametri. Ca rezultat se poate crea un singur test ce poate rula de mai multe ori - o dată pentru fiecare parametru.

Crearea unui test de parametri în JUnit 4 se face în cinci etape:

1. Crearea unui test generic fără parametri specifici.
2. Crearea metodei static, care returnează tipul Collection și marchează cu notarea @Parameter.
3. Crearea câmpurilor parametrilor clasei,ce sunt necesari pentru metoda standard descrisă în etapa 1.
4. Crearea unui constructor care leagă parametrii descriși în etapa 2, cu cîmpurile respective ale clasei, descrise în etapa 3.
5. Arătarea că execuția testelor de rulare are loc cu ajutorul clasei PARAMETRIZED.

În continuare vom lua în considerare acești pași descriși mai sus.

PASUL 1 CREAREA UNUI TEST GENERIC

Ca metodă de testare tipică se va folosi metoda sum a clasei testată CustomMath. Pentru a testa operația de adunare avem nevoie de un set alcătuit din trei parametri: două variabile și valoarea estimată a sumei. Metoda testată a clasei CustomMath este:

```
public static int sum(int x, int y){
      return x+y; // intoarce rezultatul adunarii a 2 valori
```

PASUL 2 CREAREA METODEI PENTRU INTRODUCEREA PARAMETRILOR

La această etapă, în clasa de testare se creează metoda de introducere a parametrilor, care este declarată ca statică și returnează tipul Collection . Această metodă trebuie să aibă implementată notarea @Parameters. In interiorul acestei metode este necesar să se creeze o matrice multi-dimensională Object, ulterior convertită într-o listă.

Metoda de introducere a parametrilor cu notarea @Parameters este:

```
@Parameters

    public static Collection sumValues(){

    return Arrays.asList(new Object[][]{

        {1,1,2},

        {-1,1,0},

        {10,15,25}});

    }
```

PASUL 3: CREAREA CELOR TREI CÎMPURI PENTRU SALVAREA PARAMETRILOR

```
int x,y,sumResult;
```

ETAPA 4: CREAREA CONSTRUCTORULUI

Constructorul, care este creat la această etapă îi vor fi atribuite valorile câmpurilor clasei:

```
public CustomMathTest(int x,int y, int sumResult) {

        this.x=x;

        this.y=y;

        this.sumResult=sumResult;

    }
```

ETAPA 5. EXECUȚIA CU CLASA PARAMETRIZED

Și, în sfârșit la nivelul clasei este necesar de specificat că testarea trebuie efectuată cu ajutorul parametrilor (Parameterized):

```
@RunWith (Parameterized.class)

public class CustomMathTest {
```

EXECUTAREA TESTĂRII

Acum testarea se efectuează de trei ori - în conformitate cu numărul de parametri pentru testare din metoda sumValues.

După schimbare, clasa Test va arata în modul următor:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */


import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;


/**
 *
 * @author Andrian
 */
@RunWith (Parameterized.class)
public class CustomMathTest {
     @Parameters
    public static Collection sumValues(){
    return Arrays.asList(new Object[][]{
        {1,1,2},
        {-1,1,0},
        {10,15,25}});
```

```
    }
    int x,y,sumResult;

    public CustomMathTest(int x,int y, int sumResult) {
        this.x=x;
        this.y=y;
        this.sumResult=sumResult;
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    /**
     * Test of sum method, of class CustomMath.
     */
    @Test
    public void testSum() {
        int expResult=sumResult;
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
      //  fail("The test case is a prototype.");
    }
}
```

SARCINA 3

Modificați metoda de testare testDivisionByZero (), astfel încât funcția să verifice împărțirea la zero, și de asemenea să furnizeze date de intrare corecte.

Includeți în raport varianta finală a claselor CustomMath și CustomMathTest, precum și PrtSc a rezultatului testării.

SARCINI SUPLIMENTARE

SARCINA 4

Extindeți clasa de testare, astfel încât să utilizeze metoda assertTrue și / sau assertFalse.

Adăugarea sau modificarea metodelor clasei testate includeți în raport.

CERINȚELE PRIVIND RAPORTUL LABORATORUL

Raportul trebuie să conțină:

Foaia de titlu standard.

Exercițiile 1-4 rezolvate.

Raportul este susținut în prezența raportului lucrării de laborator.